Lab 6

Taaruni Ananya | Session 008 | CRN: 15623

02/28/2025

Dr. Lan Gao

Introduction

Lab 6 is centered around practicing writing code with RISC-V as well as building on topics introduced and explored in Lab 5: Using Venus. Lab 6 looks into learning how to use tools such as VS Code and Terminal, and writing RISC-V code without the help of the same code in C. The lab explains how to use .data, str, int, sum, and main commands with ecall. The lab also gives hands-on experience with running different scenarios and reusing code to build efficient programs.

Purpose

The purpose of this lab is to continue practicing the use of Assembly code, along with the commands and practice writing RISC-V without the aid of C; this lab also works to introduce ecall within the program. This lab is meant to experiment with using the data section with defined values, loading and moving integer values, performing add, subtract, NOT, and NEG operations, and performing subtraction operations with add commands.

Main Objectives

- Learn and use VS Code
- Use data section for predefined values
- Load and move integer data in registers in assembler
- Perform add, subtract, NOT, NEG operations
- Perform a subtract operation with an add command

Brief Explanation

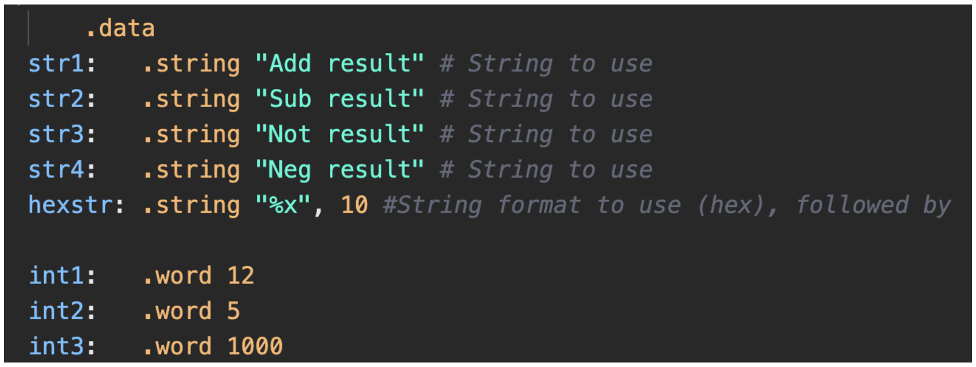
VS Code is an integrated development environment (IDE) where Venus and RISC-V can be accessed and programmed. The five parts produce different files, each with different functions in the program. This lab goes through the program as it uses predefined data values to perform various operations. This lab also makes use of Venus's ecall feature to call and complete specific tasks.

Methodology/Procedure

The procedure starts with introducing the code and loading integer values into the .data section, bringing in values with str, hexstr, int, and sum. Then, .text, .globl main, .type main, @function. Finally, the function is written in main by calling, storing, adding or subtracting, storing again, and returning the values accordingly. This process is repeated through each part and each file to perform the various operations.

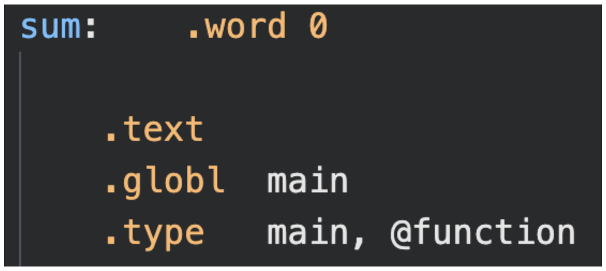
Documentation

.data Section:

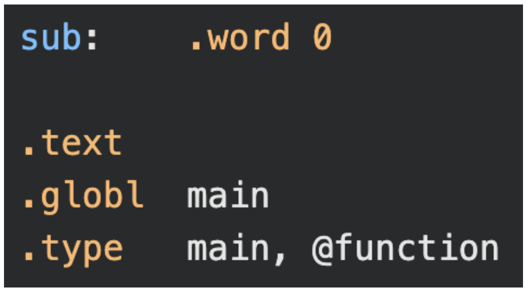


```
        .data
str1:   .string "Add result" # String to use
str2:   .string "Sub result" # String to use
str3:   .string "Not result" # String to use
str4:   .string "Neg result" # String to use
hexstr: .string "%x", 10 #String format to use (hex), followed by

int1:   .word 12
int2:   .word 5
int3:   .word 1000
```

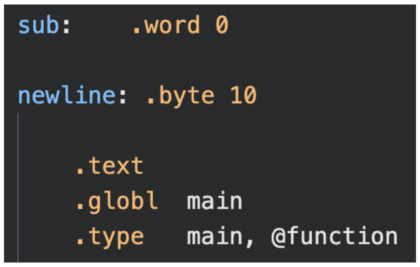
Sum/Sub/Neg/Not:



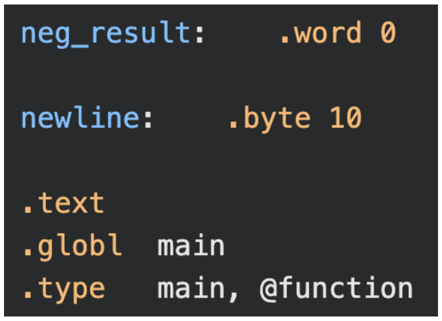
```
sum:    .word 0

    .text
    .globl  main
    .type   main, @function
```



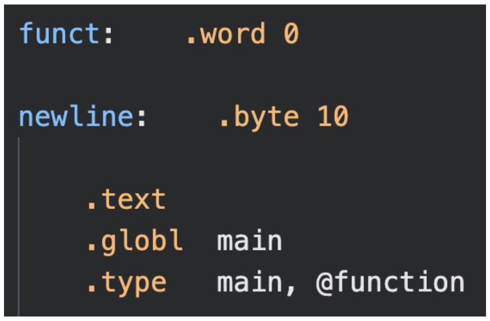
```
sub:    .word 0

.text
.globl  main
.type   main, @function
```



```
sub:    .word 0

newline: .byte 10

    .text
    .globl  main
    .type   main, @function
```



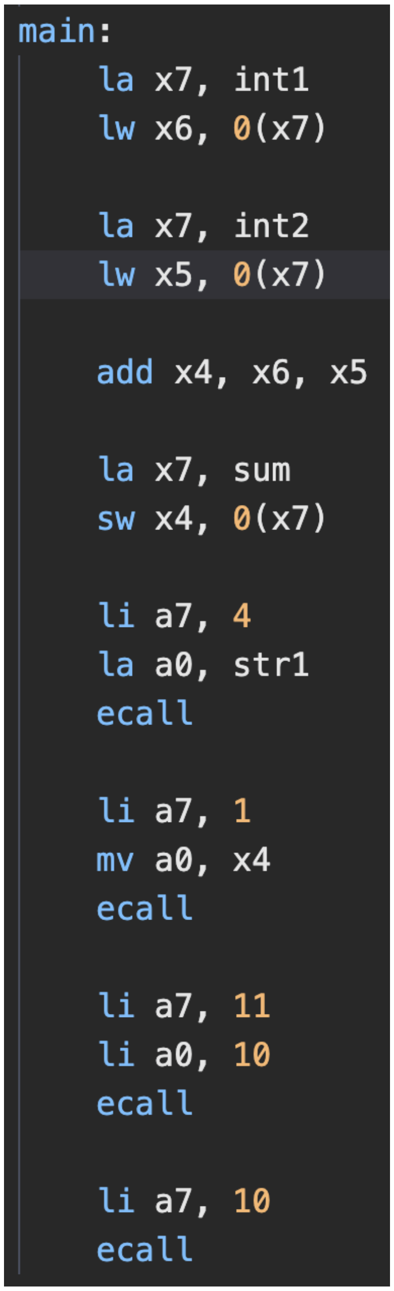
```
neg_result:    .word 0

newline:    .byte 10

.text
.globl  main
.type   main, @function
```



```
funct:    .word 0

newline:    .byte 10

    .text
    .globl  main
    .type   main, @function
```

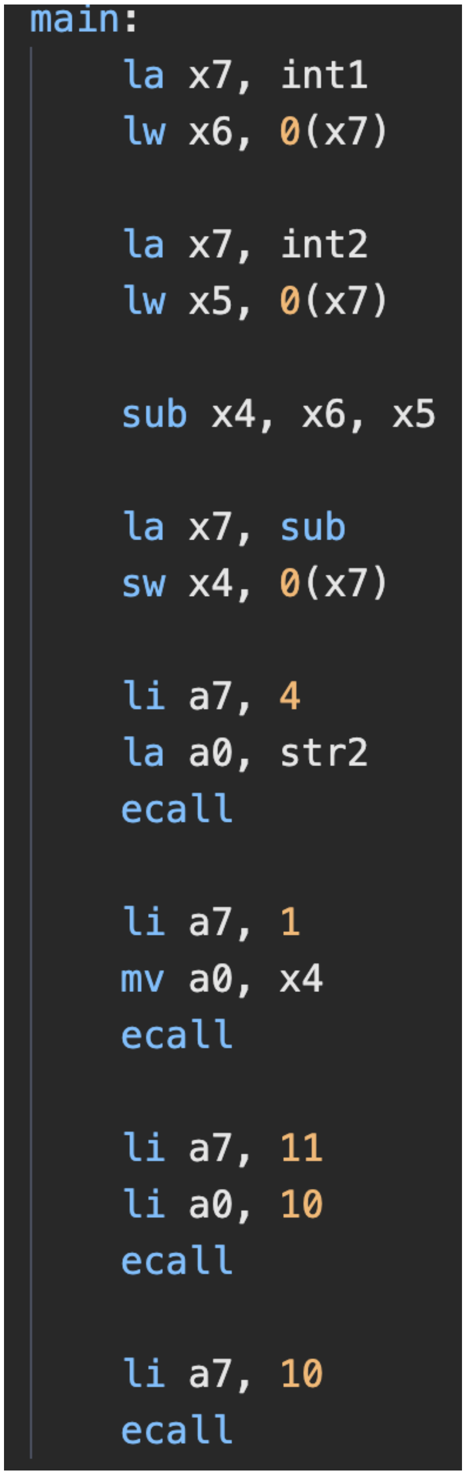
Main functions:

Part 1



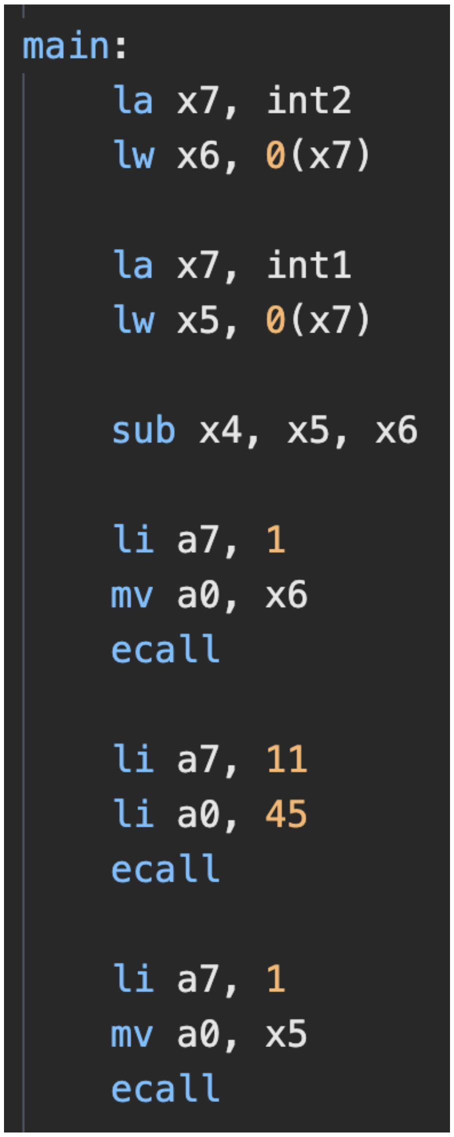
```
main:
    la x7, int1
    lw x6, 0(x7)

    la x7, int2
    lw x5, 0(x7)

    add x4, x6, x5

    la x7, sum
    sw x4, 0(x7)

    li a7, 4
    la a0, str1
    ecall

    li a7, 1
    mv a0, x4
    ecall

    li a7, 11
    li a0, 10
    ecall

    li a7, 10
    ecall
```

Part 2

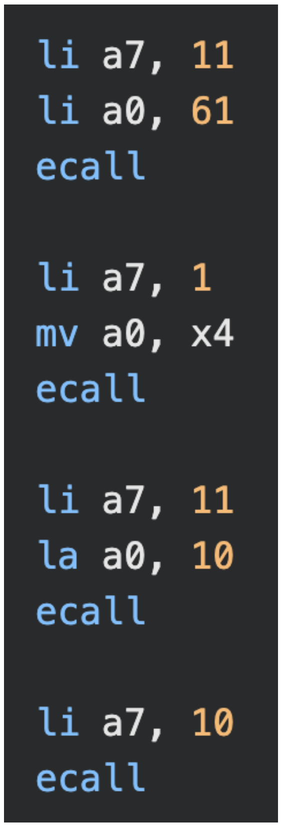


```
main:
    la x7, int1
    lw x6, 0(x7)

    la x7, int2
    lw x5, 0(x7)

    sub x4, x6, x5

    la x7, sub
    sw x4, 0(x7)

    li a7, 4
    la a0, str2
    ecall

    li a7, 1
    mv a0, x4
    ecall

    li a7, 11
    li a0, 10
    ecall

    li a7, 10
    ecall
```

Part 3:

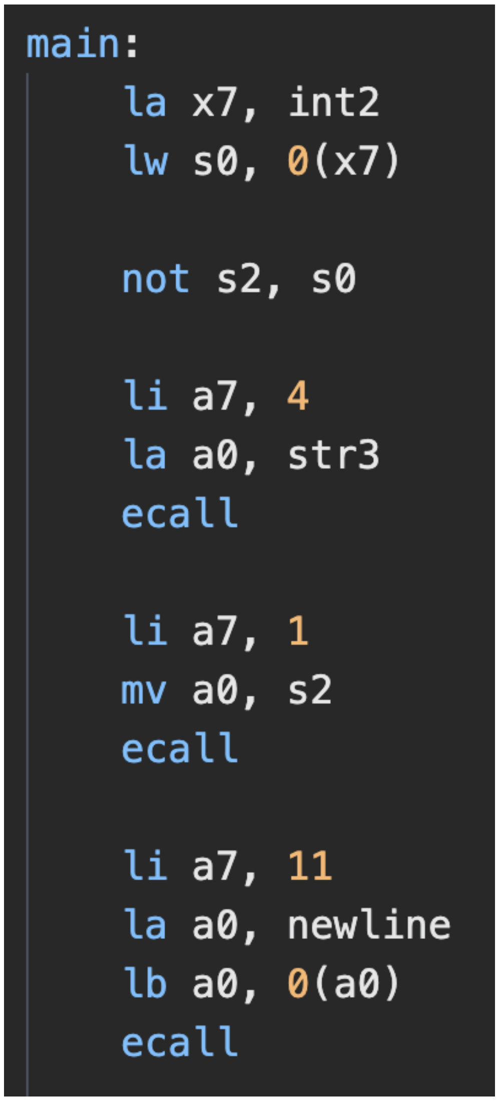


```
main:
    la x7, int2
    lw x6, 0(x7)

    la x7, int1
    lw x5, 0(x7)

    sub x4, x5, x6

    li a7, 1
    mv a0, x6
    ecall

    li a7, 11
    li a0, 45
    ecall

    li a7, 1
    mv a0, x5
    ecall
```

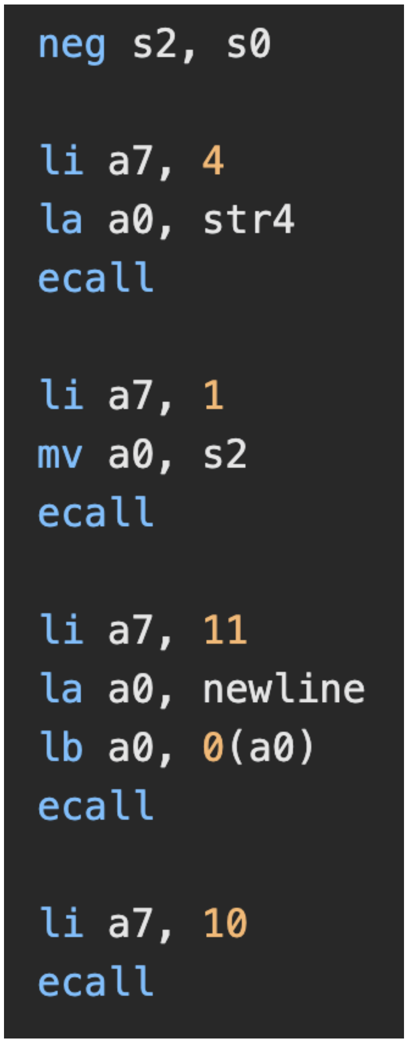


```
li a7, 11
li a0, 61
ecall

li a7, 1
mv a0, x4
ecall

li a7, 11
la a0, 10
ecall

li a7, 10
ecall
```

Part 4

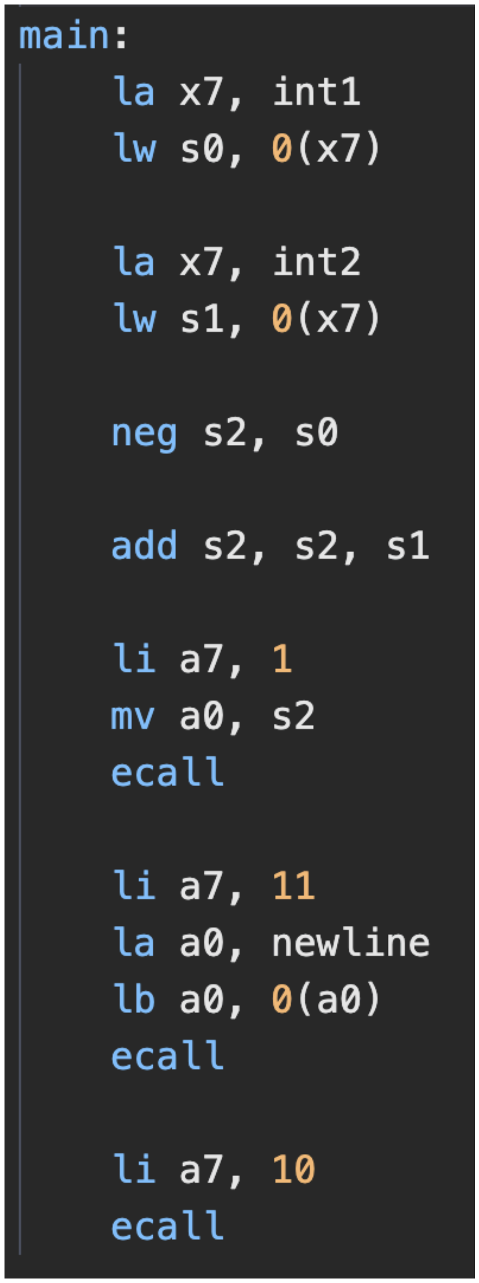


```
main:
    la x7, int2
    lw s0, 0(x7)

    not s2, s0

    li a7, 4
    la a0, str3
    ecall

    li a7, 1
    mv a0, s2
    ecall

    li a7, 11
    la a0, newline
    lb a0, 0(a0)
    ecall
```



```
neg s2, s0

li a7, 4
la a0, str4
ecall

li a7, 1
mv a0, s2
ecall

li a7, 11
la a0, newline
lb a0, 0(a0)
ecall

li a7, 10
ecall
```

Part 5



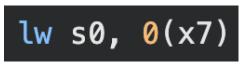
```
main:
    la x7, int1
    lw s0, 0(x7)

    la x7, int2
    lw s1, 0(x7)

    neg s2, s0

    add s2, s2, s1

    li a7, 1
    mv a0, s2
    ecall

    li a7, 11
    la a0, newline
    lb a0, 0(a0)
    ecall

    li a7, 10
    ecall
```

Questions:

Part 1:

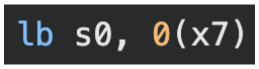
1. **What would you need to change in the program to work with bytes instead?**

   To work with bytes instead, I would need to change the load and store instructions from load word to load byte, and store word to store byte; lw to lb and sw to sb. For example:

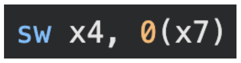


```
lw s0, 0(x7)
```

→

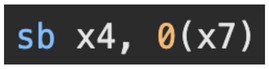


```
lb s0, 0(x7)
```



```
sw x4, 0(x7)
```

→



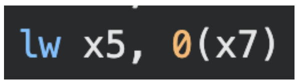
```
sb x4, 0(x7)
```

2. **What would happen if you have a large value, such as int3, and load it as a byte? Try it, then explain why the value is what it is.**

   If a large value such as int3 is loaded as a byte, the result is 232 because of the byte that is being stored in the least significant byte. The least significant byte is loaded since the value is loaded as a byte for a large value.

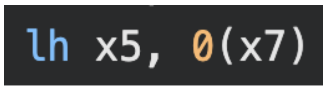
3. **What would you need to change in the program to work with halfwords instead?**

   To work with halfwords instead, I would need to change the load and store instructions from load word to load halfword, and store word to store halfword; lw to lh and sw to sh.

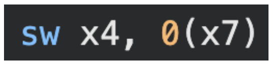


```
lw x5, 0(x7)
```

→

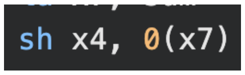


```
lh x5, 0(x7)
```



```
sw x4, 0(x7)
```

→



```
sh x4, 0(x7)
```

4. **Try using la x7, int1 followed by lw x6, 1(x7). What value do you get in x6? Why?**

   lw will be loading 4-bytes and would not be aligned as properly as it should be because int is a word with 4 bytes and loading from 1(x7) would start from a different byte–making it unaligned.

5. **Why have “sum” defined in the data section. What command(s) would we use to put the result (say, of an add command) into it?**

   In order to put the result of an add command into the “sum” in the data section, I would need to load “sum”’s address into a register and use store word to store it there.

Part 2:

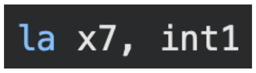
1. **Did you get the result that you expected?**

   Yes, I had to write a program to subtract the values in int1 and int2. The two corresponding values stored in these two are: 12 and 5. Subtracting them as int1 - int2, 12 – 5 results in 7. The code ran as expected to produce the expected result.

Part 3:

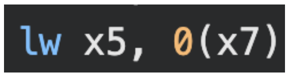
1. **How many lines of code did it take to load the value(s)?.**

   It took a total of four lines each, two for working with int1 and 2 for working with int2.

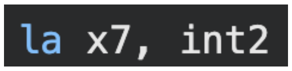


```
la x7, int1
```

&

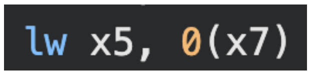


```
lw x5, 0(x7)
```



```
la x7, int2
```

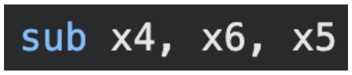
&



```
lw x5, 0(x7)
```

2. **How many lines of code did it take to do the arithmetic?**

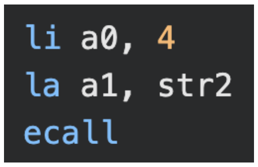
   It took one line of code to do the arithmetic, often located at the middle of the code.



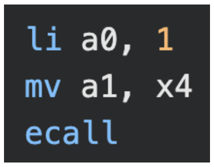
```
sub x4, x6, x5
```

3. **How many lines of code did it take to print everything?**

   It takes six lines of code to print everything, specifically located near the end of the program.



```
li a0, 4
la a1, str2
ecall
```



```
li a0, 1
mv a1, x4
ecall
```

4. **What do you observe about the code in terms of numbers of lines devoted to loading/arithmetic/printing, and repetition?**

   For loading, the code takes up only four lines of code, each of them getting and storing the values for the ints to be worked with. For arithmetic operation with subtraction, it takes only one line and is simple. Printing takes up about six lines of code in this program–more than loading and arithmetic–because it needs to get the values, load them, then return. Repetition takes up a similar amount of lines, as the values have to be called, stored, operated, etc.

Part 4:

1. **How did the computer get the result? That is, what did it do to the original value to get the value printed?**

   In terms of the NOT operation, the original value in s0 is inverted, flipping every bit. For example, from 0 to 1 and 1 to 0. In terms of the NEG operation, the value in s0 inverts all bits just like in the NOT operation but also adds 1 to the original value.

2. **Why did we skip loading a value into s1 for these commands?**

   The NOT and NEG commands are not in need of more than one operand such as s0. They store the result in s2 and don't need to load a value into s1 for these commands.

3. **Can you tell what the difference is between a simple NOT operation and 2's complement on a data?**

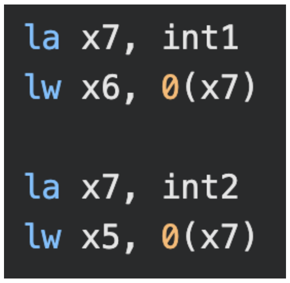
   The NOT operator works to invert each bit in all of the data,but unlike the NEG operation, it will not change the resulting number's sign. The 2's complement of NEG operation does the same as the NOT operator but adds 1 to the end result. This would change the number's sign; for example, positive to negative or negative to positive.

Part 5:

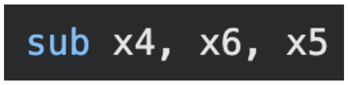
4. **We have the neg command to subtract a value from the other. Can we use the NOT operation for the same purpose? Why or why not?**

No, we can't use the NOT operation for the same purpose to subtract a value from the other because NOT only inverts all the singular bits but does not change the sign. NEG can't be replaced by NOT because it does not change the sign.

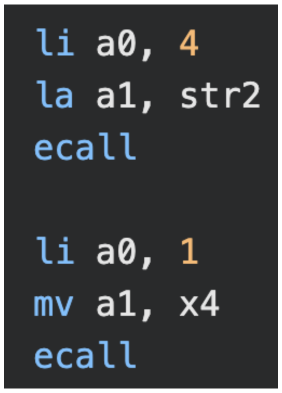
Key Code Observations



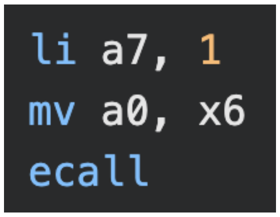
```
la x7, int1
lw x6, 0(x7)

la x7, int2
lw x5, 0(x7)
```



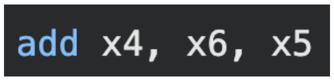
```
sub x4, x6, x5
```



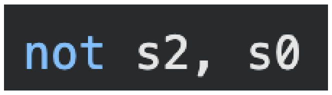
```
li a0, 4
la a1, str2
ecall

li a0, 1
mv a1, x4
ecall
```



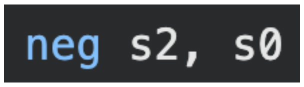
```
li a7, 1
mv a0, x6
ecall
```



```
add x4, x6, x5
```



```
not s2, s0
```



```
neg s2, s0
```